Matrix-Matrix Multiply Optimizations

David Bindel (modified by David Garmire)

September 12, 2002

1 Optimization in general

1.1 Memory hierarchy considerations

Taking advantage of the memory heirarchy is key to performance optimization on modern machines. Not only is good cache re-use important for uniprocessor performance; many of the same ideas recur in the multiprocessor world.

There are three (or maybe four) categories of cache misses:

- Compulsory (or cold start) misses: These are misses that occur when a data block is first accessed.
- Capacity misses: These are misses that occur because the cache has finite size.
- Conflict (collision) misses: These are misses that occur because the cache is not fully associative. (The *associativity* of a cache is the number of places in the cache where a block from a specific memory address can reside – in general a specific spot in memory can't just go anywhere in the cache where there is room).

The fourth "C" is coherence misses, which occur in shared-memory multiprocessors. We may say more about cache coherence later in the semester.

Why categorize cache misses in this way? One reason is that we're at Berkeley, and all our architecture courses are taught using the books by Patterson and Hennessy. A more important reason is that the model provides a useful way to understand how memory performance problems occur, and what we can do to fix them.

In the context of matrix multiplication, there isn't much we can do about compulsory misses. In more complicated computations, though, there sometimes is a way to avoid compulsory misses: recompute things! At first it seems counterintuitive to compute something and throw it away, only to compute it again at a different time (or place). If the cost of the computation is modest, though, it is often faster to recompute than to retrieve the result from main memory (or even the L2 cache). If a precomputed result is stored on disk or on a different node in a multiprocessor, recomputing can save a lot of time.

Tiling reduces capacity misses. When the working set (the data that a program uses in some given time frame) is bigger than the size of a cache, the processor will frequently have to access a larger (slower) level of memory, and performance suffers. In other words, the cache gets thrashed. If we organize matrix multiply so that the working set effectively includes even one entire matrix, it won't take long before cache thrashing sets in. By reorganizing the matrix multiply to repeatedly used data from a particular tile before moving onto the next tile, we make our working set into something that will fit more readily into cache.

Copy optimizations can help with conflict misses. If you run the provided "basic" matrix multiplication routine, you will notice some performance dips when the matrix size is a power of two. See the above figure for the timings on a Pentium 4, for example. The effect is dramatic. Because the algorithm is striding through memory by large powers of two, the elements of a particular row can only fit in a very small number of cache entries. From a performance perspective, the result is as though the cache was cut from its ordinary size down to a handful of entries (maybe just one entry in the case $n = 512$; I'm not certain what the cache sizes and associativities are on this P4).

One thing not particularly clarified by the "three Cs" model is why a stride of one is such a good thing. Caches are organized in "lines." When an L1 cache miss forces the processor to request bytes from a lower level of memory, more than just the few bytes that the processor needs are retrieved. Adjacent bytes are retrieved to fill in the rest of the cache line, on the principle that they will probably be needed soon. Since you're going to

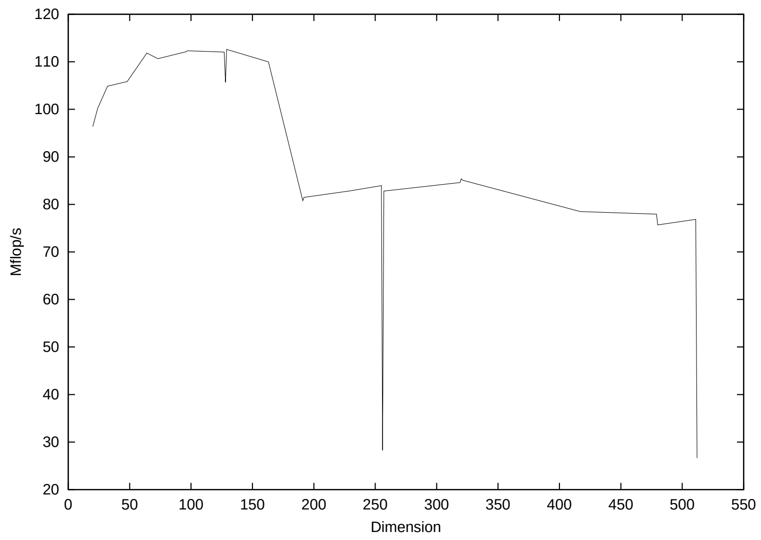

Figure 1: Naive matrix multiply times on a P4

get an entire cache line worth of material whether you want it or not, you might as well take advantage of it and use the entire line before it is discarded during a subsequent load.

1.2 Inner-loop optimization

“We should forget about small efficiencies about 97% of the time. Premature optimization is the root of all evil” – D. Knuth

Exploiting the memory hierarchy is a big part of good performance, but it isn’t everything. A modern uniprocessor provides numerous opportunities for instruction-level parallelism, and modern compilers are reasonably good about scheduling to take advantage of that parallelism. Sometimes, though, the compiler will not do the best thing. In that case, it’s up to you to show the compiler places where there are no dependencies, orders that make good use of the available functional units, etc.

When doing detailed optimization, it pays to keep a certain perspective. Making some code segment run in four cycles instead of five (or 25) is great, but if that code only makes up a small fraction of the total execution time, the effect will be negligible. In the context of matrix multiply, this means that it probably will not be particularly worthwhile to spend lots of time doing things like unrolling outer loops. *Save your effort for the places where most of your time is spent.* In the case of matrix multiply, this is the innermost loop.

Probably the simplest type of inner-loop optimization you’ll perform is loop unrolling. By unrolling a small inner loop a few times, you can hide some of the loop overhead associated with incrementing the counter and performing the branch. Unrolling too many times can actually slow your code down. Also remember that you have to write some sort of clean-up code to handle any remaining iterations if the the number of unrollings doesn’t evenly divide the total number of iterations. You will probably devise your code so that the number of loop unrollings evenly divides your tile sizes. In that case, your tile sizes will not always evenly divide the matrix size, and you will want to treat the leftover “fringes” with some care. The fringe code is a great place to commit indexing errors and other subtle bugs; it is also a place where you may lose time to sub-optimal code.

The order of floating-point instructions may also be important in determining your inner loop speed. If your code initiates a sequence of multiplies with no multiplies interspersed, you will probably lose some cycles while the processor waits for the multiplier to become available. If you mix multiplies and adds, you will probably make better use of the floating point functional units. I know that the Pentium III floating point units are pipelined, but I haven’t looked up latencies and throughputs, so I’m not certain how much of an effect you’ll see from changes in the floating point mix.

You will want to avoid unecessary data dependencies. For instance, one way to sum a sequence of numbers is to store all the partial sums in the same place:

```
a += a1
a += a2
a += a3
a += a4
```

Often, a better way to write the code would be

```
b1 = a1 + a2
b2 = a3 + a4
a = b1 + b2
```

This way, the sums `b1` and `b2` can be computed partially or completely in parallel, if the processor has multiple adders (or one pipelined adder). The compiler will probably not do this optimization for you – and indeed it *should* not generally do this optimization for you. Floating point addition is not associative (that is $a+(b+c) \neq (a+b)+c$ in the floating point world) and some numerical tricks depend on the additions being done in the specified order.

The game here is to get the instruction that produces a particular result to start a while before any instructions that use the result. That way, an instruction that takes a while to complete doesn't hold up an instruction that might use its result. In the case of matrix multiply, this is pretty easy: for instance, you could overlap the computations of two different elements of the product matrix.

1.3 Leftovers

"Lesser artists borrow. Great artists steal." – I. Stravinsky

Optimizing matrix multiplication makes a good exercise, but it probably isn't something you would care to do many times. There are good libraries available for doing numerical linear algebra (and a variety of other tasks). If you don't spend lots of time reinventing the wheel (or perhaps just reinventing steel-belted radials) you'll probably have more time to devote to the unique details of whatever problem you are contemplating.

Operations like manual loop-unrolling can be very time-consuming. It's all too easy to make a minor typo, or to copy something and forget to change the one important character, or to otherwise make a unique and frustrating mistake. I highly recommend becoming familiar with a scripting language, and writing scripts to automatically generate some of the more tedious code segments. Besides saving you typing time, it may also save some debugging time.

More generally, I highly recommend becoming familiar with the tools offered in your environment. Profilers (e.g. `gprof`) can help you find the bottlenecks in your code, so that you can spend your optimization effort wisely; debuggers (e.g. `gdb`) can help you find out what happened to cause that pesky core dump; and tools like VaMPIr on the Cray T3E can help you figure out whether you really have succeeded in overlapping communication and computation.

2 Pentium III optimization

The specific nodes you will use for the assignment are Pentium III processors, code name Katmai. See the "implementation" section of the assignment web page for information on how to specify that you want to run on those nodes. The processor has the following general characteristics:

- Clock speed is 550 MHz
- 32K non-blocking L1 cache (16K for instructions, 16K for data)
- 512K external L2 cache, running at half processor speed. The L2 cache is 4-way set associative, and is attached to the CPU by a 64 bit bus. Approximate total latency to L2 cache is 24 cycles.

There are two C compilers available on the Millennium: the GNU compiler `gcc` (version 2.96) and the Portland Group compiler `pgcc`. I have not played around much with `pgcc`, but you might be able to coax faster code out of it than out of `gcc`. It still won't save you from fiddling with tile sizes.

Information about all the `gcc` options is available in the `gcc` info page. Typing `info gcc` will get you to the page, but the `info` program has a sort of quirky interface, and it may take some time for you to figure out how to navigate. If you're familiar with `emacs`, you may find it just as easy to use the info mode. You can also do a Google search for `gcc info` and look at the first link.

`gcc` flags of interest include

- `-fargument-noalias` : Specifies that arguments do not alias one another.
- `-std=c99` : Tells the compiler to enable C99 features. In particular, you get `restrict` pointers this way. An example of how to use a `restrict` pointer is

  ```
  void foo(double * restrict A);
  ```

- `-march=pentiumpro` : Tell the compiler to use Pentium Pro specific instructions, and to schedule assuming that the code will run on a Pentium Pro.
- `-S` : Generate code to an assembly output file (extension `.s`). Sometimes looking at the "shape" of the assembly output will give you an idea what the compiler is doing (e.g. how a loop is being unrolled). You probably will not want to try stepping untangling the details of the assembly code, though. There's a reason we let compilers do that part for us.

You'll probably also want to play with the flags listed in the optimization section of the info page. Some of these flags will speed your code up; some will slow it down; and many will have no effect whatsoever.